Typing Noninterference for Reactive Programs *

Ana Almeida Matos, Gérard Boudol and Ilaria Castellani

June 7, 2004

Abstract

We propose a type system to enforce the security property of *noninterference* in a core reactive language, obtained by extending the imperative language of Volpano, Smith and Irvine with reactive primitives manipulating broadcast signals and with a form of "scheduled" parallelism. Due to the particular nature of reactive computations, the definition of noninterference has to be adapted. We give a formulation of noninterference based on bisimulation. Our type system is inspired by that introduced by Boudol and Castellani, and independently by Smith, to cope with nontermination and time leaks in a language for parallel programs with scheduling. We establish the soundness of this type system with respect to our notion of noninterference.

1 Introduction

To be widely accepted and deployed, the mobile code technology has to provide formal guarantees regarding the various security issues that it raises. For instance, foreign code should not be allowed to corrupt, or even simply to get knowledge of secret data owned by its execution context. Similarly, a supposedly trusted code should be checked for not disclosing private data to public knowledge. In [3] we have introduced a core programming model for mobile code called ULM, advocating the use of a *locally synchronous* programming style [1, 10] in a *globally asynchronous* computing context. It is therefore natural to examine the security issues from the point of view of this programming model. In this paper, we address some of these issues, and more specifically the non-disclosure property, for a simplified version of the ULM language. We recall the main features of the synchronous programming style, in its control-oriented incarnation:

Broadcast signals Program components react according to the presence or absence of signals, by computing and emitting signals that are broadcast to all components of a given "synchronous area".

Suspension Program components may be in a suspended state, because they are waiting for a signal which is absent at the moment where they get the control.

Preemption There are means to abort the execution of a program component, depending on the presence or absence of a signal.

Instants Instants are successive periods of the execution of a program, where the signals are consistently seen as present or absent by all components.

The so-called *reactive* variant of the synchronous programming style, designed by Boussinot, has been implemented in a number of languages and used for various applications, see [8, 7]. This differs from the synchronous

*Research partially funded by the EU IST FET Project MIKADO, by the french ACI Project CRISS, and by the PhD scholarship POSI/SFRH/BD/7100/2001.

language ESTEREL [2], for instance in the way absence of signals is dealt with: in reactive programming, the absence of a signal can only be determined at the end of the current instant, and reaction is postponed to the next instant. In this way, the causal paradoxes that arise in some ESTEREL programs can be avoided, making reactive programming well suited for systems where concurrent components may be dynamically added or removed, as it is the case with mobile code.

We consider here a core reactive language, which is a subset of ULM that extends the sequential language of [18] with reactive primitives and with an operator of alternating parallel composition (incorporating a fixed form of scheduling). As expected, these new constructs add expressive power to the language and induce new forms of security leaks. Moreover, the two-level nature of reactive computations, which evolve both within instants and across instants, introduces new subtleties in the definition of noninterference. We give a formulation of noninterference based on bisimulation, as is now standard [15, 14, 16, 4]. We define a type system to enforce this property of noninterference, along the lines of that proposed by Boudol and Castellani [5], and independently by Smith [16], for a language for parallel programs with scheduling. In this approach, types impose constraints on the relation between the security levels of tested and written variables and of received and emitted signals.

Let us briefly recall the intuition about noninterference: in a system with multiple security levels, information should only be allowed to flow from lower to higher (more secure) levels [9]. As usual, we assume security levels to form a lattice. However, in most of our examples, we shall use only two security levels, *low* (public, L) and *high* (secret, H). Security levels are attributed to variables and signals, using subscripts to specify them (eg. x_H is a variable of high level). In a sequential imperative language, an *insecure flow* of information, or *interference*, occurs when the initial values of high variables influence the final value of low variables. The simplest case of insecure flow is that of an assignment of the value of a high variable to a low variable, as in $y_L := x_H$. It is called *explicit (insecure) flow*. More subtle kinds of flow, called *implicit flows*, may be induced by the flow of control. An example is the program $\texttt{if}\ x_H = 0\ \texttt{then}\ y_L := 0\ \texttt{else}\ \texttt{nil}$, where at the end of execution the value of y_L may give information about x_H.

Other programs may be considered as secure or not depending on the context in which they might appear. For instance, the program

$$(\texttt{while}\ x_H \neq 0\ \texttt{do}\ \texttt{nil})\ ;\ y_L := 0 \tag{1}$$

may be considered safe in a sequential setting (since whenever it terminates it produces the same value 0 for y_L), whereas it becomes critical in the presence of parallelism or scheduling (as explained for instance in [4, 5]). When moving to a reactive setting we must reconsider the security of such programs in the new contexts.

In the ULM model we consider two kinds of parallel composition: first, there is the globally asynchronous composition of "reactive machines". This is similar to the parallel composition usually considered in the literature (see for instance [15, 5, 16]), except that it is quite natural to assume that there is no specific scheduling at this level. We do not consider this global composition here, and we expect it could be dealt with in a standard compositional manner. Then, in a locally synchronous area, that is within a reactive machine, parallel composition is quite different: like in the implementation of reactive programming [7], we assume a deterministic cooperative scheduling discipline on threads. It is well-known that scheduling induces new possibilities of flow (see [15, 14] for instance), and this is indeed the case with the sequentialisation of threads that we adopt. Consequently, programs such as (1) can be dangerous in a reactive setting. Similarly, we should question whether the reactive counterparts of the above programs pose problems. In fact they do, as we shall see in Section 3.1.

Another problem we are faced with when addressing the security of reactive programs is their ability to suspend while waiting for an absent signal, thus giving rise to a special event called *instant change*. One of the effects of an instant change is to reset all signals to "absent". With the constructs of the language we may write (for any security level) a program `pause`, whose behaviour is to suspend for the current instant, and terminate at the beginning of the next instant (see Section 2.2). This allows us to write the following program:

$$\texttt{emit}\ a_L\ ;\ \texttt{if}\ x_H = 0\ \texttt{then}\ \texttt{nil}\ \texttt{else}\ \texttt{pause} \tag{2}$$

Depending on the value of x_H, this program may either terminate within an instant, in which case a_L remains present, or suspend and change instant, in which case a_L is erased. However, since instant changes are not statically predictable in general, we do not consider as observable the change in the status of low signals that occurs in the transition from one instant to the next. Consequently, we consider (2) as safe. These considerations will lead us to adapt the definition of noninterference. We will then be able to prove that our type system is sound for this notion of noninterference.

The rest of the paper is organized as follows. In Section 2 we introduce the language and its operational semantics. Section 3 presents the type system and some properties of typed programs, including subject reduction. We then define noninterference as a bisimulation relation and prove the soundness of our type system with respect to it. Most proofs are omitted in this extended abstract. They may be found in the full paper [6].

2 The language

2.1 Syntax

We consider two infinite and disjoint sets of *variables* and *signals*, Var and Sig, ranged over by x, y, z and a, b, c respectively. We then let $Names$ be the union $Var \cup Sig$, ranged over by n, m. The set Exp of expressions includes booleans and naturals with the usual operations, but no signals. For convenience we have chosen to present the type system only in Section 3.1. However types, or more precisely *security levels*, ranged over by δ, θ, σ, already appear in the syntax of the language. Security levels constitute what we call *simple types*, and are used to type expressions and declared signals. In Section 3 we will see how more complex types for variables, signals and programs may be built from simple types.

The language of *processes* $P, Q \in Proc$ is defined by:

$$\begin{array}{rcccccccccc} P & ::= & x := e & | & \texttt{let}\ x : \delta = e\ \texttt{in}\ P & | & \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q & | & \texttt{while}\ e\ \texttt{do}\ P & | & P\,;\,Q \\ & | & \texttt{nil} & | & \texttt{emit}\ a & | & \texttt{local}\ a : \delta\ \texttt{in}\ P & | & \texttt{do}\ P\ \texttt{watching}\ a & | & \texttt{when}\ a\ \texttt{do}\ P & | & (P \curvearrowleft Q) \end{array}$$

Note the use of brackets to make the precedence of $\curvearrowleft$ unambiguous. The construct $\texttt{let}\ x : \delta = e\ \texttt{in}\ Q$ binds free occurrences of variable x in Q, whereas $\texttt{local}\ a : \delta\ \texttt{in}\ Q$ binds free occurrences of signal a in Q. The free variables and signals of a program P, noted $\texttt{fv}(P)$ and $\texttt{fs}(P)$ respectively, are defined in the usual way.

2.2 Operational Semantics

Configurations C are quadruples $\langle \Gamma, S, E, P \rangle$ composed of a type-environment Γ, a variable-store S, a signal-environment E and a program P. The type-environment is a mapping from names to the appropriate types. We denote its update by $\{x : \delta\ \textit{var}\}\Gamma$ or $\{a : \delta\ \textit{sig}\}\Gamma$, where δ *var* and δ *sig* denote types for variables and signals respectively (formally introduced in Section 3.1). A variable-store is a mapping from variables to values. By abuse of language we denote by $S(e)$ the atomic evaluation of the expression e under S, which we assume to always terminate and to produce no side effects. We denote by $\{x \mapsto S(e)\}S$ the update or extension of S with the value of e for the variable x, depending on whether the variable is present or not in the domain of S. The signal-environment is the set of signals which are considered to be present. We restrict our attention to well-formed configurations, satisfying $\texttt{fv}(P) \subseteq \texttt{dom}(S)$ and $\texttt{dom}(S) \cup E \subseteq \texttt{dom}(\Gamma)$. We will generally use the word *memory* to refer to the pair $\langle$variable-store, signal-environment$\rangle$.

A distinguishing feature of reactive programs is their ability to *suspend* while waiting for a signal. The suspension predicate, which applies to pairs of programs and signal-environments, is defined inductively by the rules in Figure 1. Suspension is introduced by the construct $\texttt{when}\ a\ \texttt{do}\ P$, in case signal a is absent. The suspension of a program P is propagated to certain contexts, namely processes of the form $P\,;\,Q$, $\texttt{do}\ P\ \texttt{watching}\ a$,

$$\text{(WHEN-SUS}_1\text{)}\ \frac{a \notin E}{(E, \texttt{when}\ a\ \texttt{do}\ P)\ddagger} \qquad \text{(WHEN-SUS}_2\text{)}\ \frac{(E,P)\ddagger}{(E, \texttt{when}\ a\ \texttt{do}\ P)\ddagger}$$

$$\text{(WATCH-SUS)}\ \frac{(E,P)\ddagger}{(E, \texttt{do}\ P\ \texttt{watching}\ a)\ddagger}$$

$$\text{(SEQ-SUS)}\ \frac{(E,P)\ddagger}{(E, P\,;\,Q)\ddagger} \qquad \text{(PAR-SUS)}\ \frac{(E,P)\ddagger \quad (E,Q)\ddagger}{(E, P \curlywedge Q)\ddagger}$$

Figure 1: Suspension predicate

$\texttt{when}\ a\ \texttt{do}\ P$ and $P \curlywedge Q$, which we call *suspendable processes*. We extend suspension to configurations by letting $\langle \Gamma, S, E, P\rangle\ddagger$ if $\langle E, P\rangle\ddagger$.

There are two forms of transitions between configurations: simple *moves*, denoted by the arrow $C \to C'$, and *instant changes*, denoted by $C \hookrightarrow C'$. These are collectively referred to as *steps*, and is denoted by $C \longmapsto C'$. The reflexive and transitive closure of these transition relations are denoted with a '$*$' as usual. An *instant* is a sequence of moves leading to termination or suspension.

2.2.1 Moves

The operational rules for imperative and reactive constructs are given in Figure 2. The functions $\texttt{newv}(N)$ and $\texttt{news}(N)$ are injective functions on finite sets of names which return a fresh name not in N, respectively a variable and a signal. They are used in order to guarantee determinism in the language. The notation $\{n/m\}P$ stands for substitution (name-capture avoiding) of m by n in P.

The imperative rules are as usual, where termination is dealt with by reduction to '`nil`'. Some comments on the reactive rules are in order. Signal emission adds a signal to the signal-environment. The local signal declaration is standard. The `watching` construct allows the execution of its body until an instant change occurs; the execution will then resume or not at the next instant depending on the presence of the signal (as explained below). As for the `when` construct, execution of its body depends on the presence of the signal; the body suspends if the signal is absent. Alternating parallel composition implements a co-routine mechanism. It executes its left component until termination or suspension, and then gives control to its right component, provided this is not already suspended.

Example 1 (Alternating parallel composition) *In this example three threads are queuing for execution in an empty signal-environment (left column). Underbraces indicate suspension. The emission of signal a by the third process unblocks the first of the suspended processes, enabling them to execute one by one and then reach termination.*

	$\{\}$	$(\underbrace{(\texttt{when}\ a\ \texttt{do}\ \texttt{emit}\ b) \curlywedge (\texttt{when}\ b\ \texttt{do}\ \texttt{emit}\ c)}) \curlywedge \texttt{emit}\ a$
$\to$	$\{\}$	$\texttt{emit}\ a \curlywedge ((\texttt{when}\ a\ \texttt{do}\ \texttt{emit}\ b) \curlywedge (\texttt{when}\ b\ \texttt{do}\ \texttt{emit}\ c))$
$\to^\star$	$\{a\}$	$(\texttt{when}\ a\ \texttt{do}\ \texttt{emit}\ b) \curlywedge (\texttt{when}\ b\ \texttt{do}\ \texttt{emit}\ c)$
$\to^\star$	$\{a,b\}$	$\texttt{when}\ b\ \texttt{do}\ \texttt{emit}\ c$
$\to^\star$	$\{a,b,c\}$	$\texttt{nil}$

We have seen that suspension of a thread may be lifted during an instant upon emission of the signal by another thread in the pool. This is no longer possible in a program in which all threads are suspended. When this situation is reached, an instant change occurs.

(ASSIGN-OP) $\langle \Gamma, S, E, x := e \rangle \to \langle \Gamma, \{x \mapsto S(e)\}S, E, \texttt{nil} \rangle$

(SEQ-OP$_1$) $\langle \Gamma, S, E, \texttt{nil}\,;\, Q \rangle \to \langle \Gamma, S, E, Q \rangle$

(SEQ-OP$_2$) $$\frac{\langle \Gamma, S, E, P \rangle \to \langle \Gamma', S', E', P' \rangle}{\langle \Gamma, S, E, P\,;\, Q \rangle \to \langle \Gamma', S', E', P'\,;\, Q \rangle}$$

(LET-OP) $$\frac{y = \texttt{newv}(\texttt{dom}(\Gamma))}{\langle \Gamma, S, E, \texttt{let}\ x : \delta = e\ \texttt{in}\ P \rangle \to \langle \{y : \delta\ \textit{var}\}\Gamma, \{y \mapsto S(e)\}S, E, \{y/x\}P \rangle}$$

(COND-OP$_1$) $$\frac{S(e) = \texttt{true}}{\langle \Gamma, S, E, \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \rangle \to \langle \Gamma, S, E, P \rangle}$$

(COND-OP$_2$) $$\frac{S(e) = \texttt{false}}{\langle \Gamma, S, E, \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \rangle \to \langle \Gamma, S, E, Q \rangle}$$

(WHILE-OP$_1$) $$\frac{S(e) = \texttt{true}}{\langle \Gamma, S, E, \texttt{while}\ e\ \texttt{do}\ P \rangle \to \langle \Gamma, S, E, P\,;\, \texttt{while}\ e\ \texttt{do}\ P \rangle}$$

(WHILE-OP$_2$) $$\frac{S(e) = \texttt{false}}{\langle \Gamma, S, E, \texttt{while}\ e\ \texttt{do}\ P \rangle \to \langle \Gamma, S, E, \texttt{nil} \rangle}$$

(EMIT-OP) $\langle \Gamma, S, E, \texttt{emit}\ a \rangle \to \langle \Gamma, S, \{a\} \cup E, \texttt{nil} \rangle$

(LOCAL-OP) $$\frac{b = \texttt{news}(\texttt{dom}(\Gamma))}{\langle \Gamma, S, E, \texttt{local}\ a : \delta\ \texttt{in}\ P \rangle \to \langle \{b : \delta\ \textit{sig}\}\Gamma, S, E, \{b/a\}P \rangle}$$

(WATCH-OP$_1$) $\langle \Gamma, S, E, \texttt{do}\ \texttt{nil}\ \texttt{watching}\ a \rangle \to \langle \Gamma, S, E, \texttt{nil} \rangle$

(WATCH-OP$_2$) $$\frac{\langle \Gamma, S, E, P \rangle \to \langle \Gamma', S', E', P' \rangle}{\langle \Gamma, S, E, \texttt{do}\ P\ \texttt{watching}\ a \rangle \to \langle \Gamma', S', E', \texttt{do}\ P'\ \texttt{watching}\ a \rangle}$$

(WHEN-OP$_1$) $$\frac{a \in E}{\langle \Gamma, S, E, \texttt{when}\ a\ \texttt{do}\ \texttt{nil} \rangle \to \langle \Gamma, S, E, \texttt{nil} \rangle}$$

(WHEN-OP$_2$) $$\frac{a \in E \quad \langle \Gamma, S, E, P \rangle \to \langle \Gamma', S', E', P' \rangle}{\langle \Gamma, S, E, \texttt{when}\ a\ \texttt{do}\ P \rangle \to \langle \Gamma', S', E', \texttt{when}\ a\ \texttt{do}\ P' \rangle}$$

(PAR-OP$_1$) $\langle \Gamma, S, E, \texttt{nil} \upharpoonleft Q \rangle \to \langle \Gamma, S, E, Q \rangle$

(PAR-OP$_2$) $$\frac{\langle \Gamma, S, E, P \rangle \to \langle \Gamma', S', E', P' \rangle}{\langle \Gamma, S, E, P \upharpoonleft Q \rangle \to \langle \Gamma', S', E', P' \upharpoonleft Q \rangle}$$

(PAR-OP$_3$) $$\frac{\langle E, P \rangle\ddagger \quad \neg\langle E, Q \rangle\ddagger}{\langle \Gamma, S, E, P \upharpoonleft Q \rangle \to \langle \Gamma, S, E, Q \upharpoonleft P \rangle}$$

Figure 2: Operational semantics of moves

$$\text{(INSTANT-OP)}\quad \frac{\langle E, P\rangle \ddagger}{\langle \Gamma, S, E, P\rangle \hookrightarrow \langle \Gamma, S, \emptyset, \lfloor P \rfloor_E\rangle}$$

$$\lfloor \texttt{do}\ P\ \texttt{watching}\ a \rfloor_E \stackrel{\text{def}}{=} \begin{cases} \texttt{nil} & \text{if } a \in E \\ \texttt{do}\ \lfloor P \rfloor_E\ \texttt{watching}\ a & \text{otherwise} \end{cases} \qquad \lfloor P\,;\, Q \rfloor_E \stackrel{\text{def}}{=} \lfloor P \rfloor_E\,;\, Q$$

$$\lfloor \texttt{when}\ a\ \texttt{do}\ P \rfloor_E \stackrel{\text{def}}{=} \begin{cases} \texttt{when}\ a\ \texttt{do}\ \lfloor P \rfloor_E & \text{if } a \in E \\ \texttt{when}\ a\ \texttt{do}\ P & \text{otherwise} \end{cases} \qquad \lfloor P \curlywedge Q \rfloor_E \stackrel{\text{def}}{=} \lfloor P \rfloor_E \curlywedge \lfloor Q \rfloor_E$$

Figure 3: Operational semantics of instant changes

2.2.2 Instant changes

Suspension of a configuration marks the end of an instant. At this point, all signals are reset to absent (the new signal-environment is the empty set) and all the suspended subprocesses of the form $\texttt{do}\ P\ \texttt{watching}\ a$ whose watched signal is present are killed. Indeed, the `watching` construct provides a way to recover from the deadlock situation. The semantics of *instant changes* is defined in Figure 3. The function $\lfloor P \rfloor_E$ is meant to be applied to suspended processes (see Figure 1) and therefore is only defined for them.

Instant changes are programmable; we hinted in the Introduction the possibility of encoding a primitive that enforces suspension of a thread until instant change. This primitive, which we call `pause`, is defined as follows.

Example 2 (`pause`) *Here the local declaration of signal a ensures that the signal cannot be emitted outside the scope of its declaration, and therefore that the program will suspend. At this point, the presence of b is checked, and since it has been emitted, the subprogram* $(\texttt{when}\ a\ \texttt{do}\ \texttt{nil})$*, where a is replaced by a fresh variable a', is aborted (i.e. turned into* `nil`*) at the beginning of the next instant.*

$$\begin{array}{lll} & \{\} & \texttt{local}\ a : \delta\ \texttt{in}\ (\texttt{local}\ b : \theta\ \texttt{in}\ (\texttt{emit}\ b\,;\, \texttt{do}\ (\texttt{when}\ a\ \texttt{do}\ \texttt{nil})\ \texttt{watching}\ b)) \\ \rightarrow^\star & \{\} & \texttt{emit}\ b\,;\, \texttt{do}\ (\texttt{when}\ a'\ \texttt{do}\ \texttt{nil})\ \texttt{watching}\ b' \\ \rightarrow^\star & \{b\} & \underbrace{\texttt{do}\ (\underbrace{\texttt{when}\ a'\ \texttt{do}\ \texttt{nil}}\ \texttt{watching}\ b')} \\ \hookrightarrow & \{\} & \texttt{nil} \end{array}$$

The following is a program where an instant change breaks a causality cycle:

$$\texttt{emit}\ a\,;\, ((\texttt{when}\ b\ \texttt{do}\ \texttt{emit}\ c) \curlywedge (\texttt{do}\ (\texttt{when}\ c\ \texttt{do}\ \texttt{emit}\ b)\ \texttt{watching}\ a)\,;\, \texttt{emit}\ b)$$

Here the whole program suspends after the emission of a. Then, since a is present, the body of the `watching` construct is killed and a new instant starts, during which b is emitted, thus unblocking the other thread and allowing c to be emitted.

2.2.3 Execution paths

A computation of a configuration has the form:

$$\langle \Gamma, S, E, P\rangle \rightarrow^\star \langle \Gamma_1, S_1, E_1, P_1\rangle \hookrightarrow \langle \Gamma_1, S_1, \emptyset, \lfloor P_1 \rfloor_{E_1}\rangle \rightarrow^\star \langle \Gamma_2, S_2, E_2, P_2\rangle \hookrightarrow \ldots$$

One can check that every configuration is able to perform a step if and only if its program is different from `nil`. The form of this step, simple move or instant change, depends on whether P is suspended or not. Moreover, the following result establishes the uniqueness of this step.

Theorem 2.1 (Determinism) *Any configuration $C = \langle \Gamma, S, E, P\rangle$ is in exactly one of three states:* terminated, *if $P = \texttt{nil}$;* suspended, *if $\langle E, P\rangle\ddagger$, in which case $\exists! C' : C \hookrightarrow C'$;* active, *in which case $\exists! C' : C \to C'$.*

3 Noninterference

3.1 Type System

We now introduce our type system, whose role is to rule out insecure programs. In the Introduction we illustrated the notion of implicit flow in sequential programs. Reactive constructs should also forbid "low writes" after "high tests", as in $\texttt{when}\ a_H\ \texttt{do}\ \texttt{emit}\ b_L$. To see this, consider the program

$$\texttt{emit}\ c_L\,;\ (\texttt{do}\ (\texttt{when}\ a_H\ \texttt{do}\ \texttt{emit}\ b_L)\ \texttt{watching}\ c_L) \tag{3}$$

Whether a_H is present or not this program always terminates (in one or two instants respectively), emitting b_L only if a_H is present. Also the more subtle program (1) has its reactive counterpart $(\texttt{when}\ a_H\ \texttt{do}\ \texttt{nil})\,;\ \texttt{emit}\ b_L$. Again, this could be viewed as safe when run in isolation. However, when composed with other threads, this program can be source of interferences as suspension can be lifted by the emission of signal a_H. Consider for instance the program $\gamma \lhd (\alpha \lhd \beta)$ (which is the reactive analogue of the `PIN` example of [15, 5]), where

$$\begin{aligned}\gamma &: \texttt{if}\ \texttt{PIN}_H = 0\ \texttt{then}\ \texttt{emit}\ a_H\ \texttt{else}\ \texttt{emit}\ b_H\\ \alpha &: \texttt{when}\ a_H\ \texttt{do}\ \texttt{nil}\,;\ \texttt{emit}\ c_L\,;\ \texttt{emit}\ b_H\\ \beta &: \texttt{when}\ b_H\ \texttt{do}\ \texttt{nil}\,;\ \texttt{emit}\ d_L\,;\ \texttt{emit}\ a_H\end{aligned}$$

If $\texttt{PIN}_H = 0$, no suspension occurs: α is executed before β, and c_L is emitted before d_L. If $\texttt{PIN}_H \neq 0$, α is initially suspended and β is executed first, emitting d_L and then unblocking α. In this case c_L is emitted after d_L.

Note that imperative constructs with high tests, followed by "low writes" (as in the imperative program (1) of the Introduction) remain problematic in a reactive setting, for we can write the program $\gamma' \lhd (\alpha' \lhd \beta')$:

$$\begin{aligned}\gamma' &: \texttt{if}\ \texttt{PIN}_H = 0\ \texttt{then}\ x_H := 0\ \texttt{else}\ x_H := 1\\ \alpha' &: (\texttt{while}\ x_H = 0\ \texttt{do}\ \texttt{pause}\,;\ r_L := 0\,;\ x_H := 0)\\ \beta' &: (\texttt{while}\ x_H = 1\ \texttt{do}\ \texttt{pause}\,;\ r_L := 1\,;\ x_H := 1)\end{aligned}$$

Reactive concurrency introduces new leaks that are not exhibited with the usual asynchronous concurrency. Consider the program $(\gamma'' \lhd \alpha'') \lhd \beta''$, running in two different environments, $E_1 = \{a_H, z_H\}$ and $E_2 = \{z_H\}$:

$$\begin{aligned}\gamma'' &: (\texttt{pause}\,;\ x_L := 1)\\ \alpha'' &: (\texttt{do}\ (\texttt{when}\ a_H\ \texttt{do}\ \texttt{nil})\ \texttt{watching}\ z_H \lhd \texttt{when}\ b_L\ \texttt{do}\ x_L := 0)\\ \beta'' &: (\texttt{nil}\,;\ \texttt{pause}\,;\ \texttt{emit}\ b_L)\end{aligned}$$

The threads γ'' and α'' are running in parallel, containing different low assignments on x_L. Notice that while γ'' starts by suspending itself, the thread α'' suspends if and only if signal a_H is absent. Therefore, only in an environment where a_H is present, will γ'' and α'' start by switching positions. The `nil` component in β'' (although technically redundant) stresses that this thread gains control before suspension, thus ensuring that the first thing that happens after the change of instant is the emission of signal b_L. What remains to be done is the two assignments to the low variable x_L, but the order in which they will happen depends in the order of appearance of the continuations of γ'' and α'' (which as we've seen depends on the the presence of the high signal a_H). This example suggests that it is possible to make the order of execution of neighboring threads depend on high signals. The tree of threads can then be seen as the body of a conditional, where any high test is its potential guard. This motivates the introduction of some extra conditions in the typing of reactive concurrency, similar to those used for conditionals in [5, 16].

Let us now present our type system. As we mentioned in Section 2, expressions will be typed with *simple types*, which are just security levels δ, θ, σ. As usual, these are assumed to form a lattice $(\mathcal{T}, \leq)$, where the order relation $\leq$ stands for "less secret than" and $\wedge, \vee$ denote meet and join. Starting from simple types we build *variable types* of the form δ *var* and *signal types* of the form δ *sig*. Program types will have the form (θ, σ) *cmd*, as in [5, 16]. Here the first component θ represents a lower bound on the level of written variables and emitted signals, while the second component σ is an upper bound on the level of tested variables and signals.

$$\text{(NIL)} \quad \Gamma \vdash \texttt{nil} : (\theta, \sigma)\ \textit{cmd}$$

$$\text{(ASSIGN)} \quad \frac{\Gamma \vdash e : \theta \quad \Gamma(x) = \theta\ \textit{var}}{\Gamma \vdash x := e : (\theta, \sigma)\ \textit{cmd}}$$

$$\text{(LET)} \quad \frac{\Gamma \vdash e : \delta \qquad \{x : \delta\ \textit{var}\}\Gamma \vdash P : (\theta, \sigma)\ \textit{cmd}}{\Gamma \vdash \texttt{let}\ x : \delta = e\ \texttt{in}\ P : (\theta, \sigma)\ \textit{cmd}}$$

$$\text{(SEQ)} \quad \frac{\Gamma \vdash Q_1 : (\theta_1, \sigma_1)\ \textit{cmd} \quad \Gamma \vdash Q_2 : (\theta_2, \sigma_2)\ \textit{cmd} \quad \sigma_1 \leq \theta_2}{\Gamma \vdash Q_1\,;\, Q_2 : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2)\ \textit{cmd}}$$

$$\text{(COND)} \quad \frac{\Gamma \vdash e : \delta \quad \Gamma \vdash P : (\theta, \sigma)\ \textit{cmd} \quad \Gamma \vdash Q : (\theta, \sigma)\ \textit{cmd} \quad \delta \leq \theta}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q : (\theta, \delta \vee \sigma)\ \textit{cmd}}$$

$$\text{(WHILE)} \quad \frac{\Gamma \vdash e : \delta \quad \Gamma \vdash P : (\theta, \sigma)\ \textit{cmd} \quad \delta \vee \sigma \leq \theta}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ P : (\theta, \delta \vee \sigma)\ \textit{cmd}}$$

$$\text{(EMIT)} \quad \frac{\Gamma(a) = \theta\ \textit{sig}}{\Gamma \vdash \texttt{emit}\ a : (\theta, \sigma)\ \textit{cmd}}$$

$$\text{(LOCAL)} \quad \frac{\{a : \delta\ \textit{sig}\}\Gamma \vdash P : (\theta, \sigma)\ \textit{cmd}}{\Gamma \vdash \texttt{local}\ a : \delta\ \texttt{in}\ P : (\theta, \sigma)\ \textit{cmd}}$$

$$\text{(WATCH)} \quad \frac{\Gamma(a) = \delta\ \textit{sig} \quad \Gamma \vdash P : (\theta, \sigma)\ \textit{cmd} \quad \delta \leq \theta}{\Gamma \vdash \texttt{do}\ P\ \texttt{watching}\ a : (\theta, \delta \vee \sigma)\ \textit{cmd}}$$

$$\text{(WHEN)} \quad \frac{\Gamma(a) = \delta\ \textit{sig} \quad \Gamma \vdash P : (\theta, \sigma)\ \textit{cmd} \quad \delta \leq \theta}{\Gamma \vdash \texttt{when}\ a\ \texttt{do}\ P : (\theta, \delta \vee \sigma)\ \textit{cmd}}$$

$$\text{(PAR)} \quad \frac{\Gamma \vdash P : (\theta_1, \sigma_1)\ \textit{cmd} \quad \Gamma \vdash Q : (\theta_2, \sigma_2)\ \textit{cmd} \quad \sigma_1 \leq \theta_2 \quad \sigma_2 \leq \theta_1}{\Gamma \vdash P \urcorner Q : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2)\ \textit{cmd}}$$

$$\text{(SUB)} \quad \frac{\Gamma \vdash P : (\theta, \sigma)\ \textit{cmd} \quad \theta \geq \theta' \quad \sigma \leq \sigma'}{\Gamma \vdash P : (\theta', \sigma')\ \textit{cmd}}$$

$$\text{(EXPR)} \quad \frac{\forall x_i \in \texttt{fv}(e).\delta \geq \theta_i \text{ where } \Gamma(x_i) = \theta_i\ \textit{var}}{\Gamma \vdash e : \delta}$$

Figure 4: Typing Rules

Our type system is presented in Figure 4. It is analogous to the one in [5, 16], apart from the new restrictions on parallel composition. The types for the `when` and `watching` commands are similar to those for the `while` command, since their semantics also consists of the execution of a process under a guard. As regards reactive parallel composition, the side conditions express the fact that any high test is a potential guard to the order of execution of concurrent threads. It forbids a thread from performing assignments and emissions at levels that are not higher than or equal to those of the tested variables and signals of parallel threads.

One may notice that these side conditions restrict the compositionality of the type system and introduce some

overhead (two comparisons of security levels) when adding new threads in the system. This is the price we pay for allowing loops with high guards such as $\texttt{while}\ x_H = 0\ \texttt{do}\ \texttt{nil}$ (which are rejected by previous type systems, as [15, 14]) in the context of a co-routine mechanism. However, it might be worth examining if this restriction could be lifted to some extent by means of techniques proposed for other concurrent languages ([11, 12]).

3.2 Properties of typed programs

It is easy to see from the semantic rules that computation may affect a type-environment only by extending it with fresh names. Our first result states that types are preserved along execution.

Theorem 3.1 (Subject Reduction) *If $\Gamma \vdash P : (\theta, \sigma)$ cmd and $\langle \Gamma, S, E, P\rangle \longmapsto \langle \Gamma', S', E', P'\rangle$ then $\Gamma' \vdash P' : (\theta, \sigma)$ cmd.*

Our next result ensures that program types have the intended properties. We use the generic term "guard" for either a tested variable or a tested signal.

Proposition 3.2 (Guard Safety and Confinement)

1. *If $\Gamma \vdash P : (\theta, \sigma)$ cmd then every guard in P has type $\delta \leq \sigma$.*
2. *If $\Gamma \vdash P : (\theta, \sigma)$ cmd then every variable assigned to in P and every signal emitted in P has security level δ (that is type δ var or δ sig, respectively) with $\theta \leq \delta$.*

We now introduce some terminology that will be useful to define our notion of indistinguishability. We use $\mathcal{L}$ to designate a *downward-closed set of security levels*, that is a set $\mathcal{L} \subseteq \mathcal{T}$ satisfying $\theta \in \mathcal{L}\ \&\ \sigma \leq \theta \Rightarrow \sigma \in \mathcal{L}$. The *low memory* is the portion of the variable-store and signal-environment to which the type-environment associates "low security levels" (i.e. security levels in $\mathcal{L}$). Two memories are said to be low-equal if their low parts coincide:

Definition 3.1 ($\mathcal{L}, \Gamma$-equality of Memories and Configurations) $\langle S, E\rangle =^{\Gamma}_{\mathcal{L}} \langle R, F\rangle \stackrel{def}{\Leftrightarrow}$
$\forall x.\Gamma(x) = \theta\ \textit{var}\ \&\ \theta \in \mathcal{L} \Rightarrow S(x) = R(x)$ *and* $\forall a.\Gamma(a) = \theta\ \textit{sig}\ \&\ \theta \in \mathcal{L} \Rightarrow a \in E \Leftrightarrow a \in F$. *Two configurations are said to be low-equal when they have low-equal memories.*

There is a class of programs for which the security property is trivial to establish because of their inability to change low memory. As usual, we will refer to these as *high programs*. Here we distinguish two classes of high programs based on a syntactic, respectively semantic, analysis:

Definition 3.2 (High Reactive Programs)

1. Syntactically high programs *$\mathcal{H}^{\Gamma,\mathcal{L}}_{\mathtt{syn}}$ is inductively defined by: $P \in \mathcal{H}^{\Gamma,\mathcal{L}}_{\mathtt{syn}}$ if*

- $P = (x := e)$, *and* $\Gamma(x) = \theta\ \textit{var} \Rightarrow \theta \notin \mathcal{L}$.
- $P = (\texttt{emit}\ a)$, *and* $\Gamma(x) = \theta\ \textit{sig} \Rightarrow \theta \notin \mathcal{L}$.
- $P = \texttt{let}\ x : \delta = e\ \texttt{in}\ Q$, *and* $Q \in \mathcal{H}^{\Gamma \cup \{x:\delta\ var\},\mathcal{L}}_{\mathtt{syn}}$, *and*
- $P = \texttt{local}\ a : \delta\ \texttt{in}\ Q$, *and* $Q \in \mathcal{H}^{\Gamma \cup \{a:\delta\ sig\},\mathcal{L}}_{\mathtt{syn}}$, *and*
- $P = (Q_1\ ;\ Q_2)$, $P = (\texttt{if}\ e\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2)$, $P = (\texttt{while}\ e\ \texttt{do}\ Q_1)$, $P = (\texttt{when}\ a\ \texttt{do}\ Q_1)$, $P = (\texttt{do}\ Q_1\ \texttt{watching}\ a)$, *or* $P = (Q_1 \nwarrow Q_2)$, *where* $Q_i \in \mathcal{H}^{\Gamma,\mathcal{L}}_{\mathtt{syn}}$ *for* $i = 1, 2$.

2. Semantically high programs *$\mathcal{H}^{\Gamma,\mathcal{L}}_{\mathtt{sem}}$ is coinductively defined by $P \in \mathcal{H}^{\Gamma,\mathcal{L}}_{\mathtt{sem}}$ implies:*

- $\forall S, E, \langle \Gamma, S, E, P\rangle \rightarrow \langle \Gamma', S', E', P'\rangle$ *implies* $\langle S, E\rangle =^{\Gamma}_{\mathcal{L}} \langle S', E'\rangle$ *and* $P' \in \mathcal{H}^{\Gamma',\mathcal{L}}_{\mathtt{sem}}$, *and*
- $\forall S, E, \langle \Gamma, S, E, P\rangle \hookrightarrow \langle \Gamma', S', E', P'\rangle$ *implies* $P' \in \mathcal{H}^{\Gamma',\mathcal{L}}_{\mathtt{sem}}$

Note that $P = \texttt{let}\ x : \delta = e\ \texttt{in}\ Q$ (as well as $P = \texttt{local}\ a : \delta\ \texttt{in}\ Q$) is considered syntactically high even if $\delta \notin \mathcal{L}$, provided Q is syntactically high in the extended typing environment. It may be shown that both properties are preserved by execution. As argued in the Introduction, we do not consider as relevant the initialization of the low signal environment that is induced by instant changes. This is reflected by the absence of the low equality condition after instant changes in Definition 3.2.2 (recall that the variable-store S is not modified during an instant change). Since instant changes are not statically predictable, this assumption allows us to deduce, from the syntactic property of being a high program, the corresponding behavioural property. In other words, the set of syntactically high programs is a subset of the semantically high ones, that is $\mathcal{H}^{\Gamma,\mathcal{L}}_{\mathsf{syn}} \subseteq \mathcal{H}^{\Gamma,\mathcal{L}}_{\mathsf{sem}}$. An example of a semantically high program that is not syntactically high is $\texttt{if true then nil else}\ y_L := 0$.

The key result for proving noninterference is the following theorem, whose proof is quite elaborate and therefore omitted here (it uses the notions of $\mathcal{L}$-boundedness and $\mathcal{L}$-guardedness as in [5] and some intermediate results). This result states that typed programs which immediately fork because of a high test, are syntactically high. Therefore, we are sure that the only changes in low memory occurring beyond such a fork occur at instant changes, and do not involve variables.

Theorem 3.3 (Forking programs) *Let P be typable in Γ, $\langle S, E\rangle =^{\Gamma}_{\mathcal{L}} \langle R, F\rangle$, $\langle \Gamma, S, E, P\rangle \longmapsto \langle \Gamma_1, S_1, E_1, P_1\rangle$, $\langle \Gamma, R, F, P\rangle \longmapsto \langle \Gamma_2, R_2, F_2, P_2\rangle$, and $\langle S_1, E_1\rangle \neq^{\Gamma_1\cap\Gamma_2}_{\mathcal{L}} \langle R_2, F_2\rangle$ or $\langle \Gamma_1, P_1\rangle \neq \langle \Gamma_2, P_2\rangle$ or $\langle E, P\rangle\ddagger$ iff $\neg\langle F, P\rangle\ddagger$. Then $P \in \mathcal{H}^{\Gamma,\mathcal{L}}_{\mathsf{syn}}$. ([1])*

This is quite a strong result, for it says that if there is any difference in the first step of two computations of a typable program under low-equal memories, then the program code contains no low-assignments or emissions. The intuition is that such a difference reflects the passage of a high test, and our type system guarantees that no changes in low memory will follow. Since the type system is based on a syntactical analysis, the result uses the syntactic notion of high program. For an example of a forking program which becomes syntactically high (after one step of computation) see (2) in the Introduction.

3.3 Security notion and soundness of the type system

In this section we define reactive bisimulation, and prove our noninterference result with respect to it. The two-level nature of reactive computations, together with the asymmetry in signal removal and emission, poses some challenges in the definition of noninterference. However, these subtleties can be factored out through the notion of high program.

Definition 3.3 ($\mathcal{L}$-Bisimulation equivalence ($\approx_{\mathcal{L}}$)) *The equivalence $\approx_{\mathcal{L}}$ is the largest symmetric relation $\mathcal{R}$ such that $C_1 \mathcal{R} C_2$, where $C_1 = \langle \Gamma_1, S_1, E_1, P_1\rangle$ and $C_2 = \langle \Gamma_2, S_2, E_2, P_2\rangle$, imply:*

- $C_1 =^{\Gamma_1\cap\Gamma_2}_{\mathcal{L}} C_2$, *and*
- *either*
  - *a.* $P_i \in \mathcal{H}^{\Gamma_i,\mathcal{L}}_{\mathsf{sem}}$ *for* $i = 1, 2$, *or*
  - *b.* $C_1 \longmapsto C_1'$ *implies* $\exists C_2'$ *such that* $C_2 \longmapsto^{\star} C_2'$ *and* $C_1' \mathcal{R} C_2'$

We can now formalize the notion of *secure program* in the usual way:

Definition 3.4 (Γ-Secure Programs) *P is secure in the typing context Γ if for any downward-closed set $\mathcal{L}$ of security levels and for any S_1, E_1, S_2, E_2 such that $\langle S_1, E_1\rangle =^{\Gamma}_{\mathcal{L}} \langle S_2, E_2\rangle$ then $\langle \Gamma, S_1, E_1, P\rangle \approx_{\mathcal{L}} \langle \Gamma, S_2, E_2, P\rangle$.*

Finally, we are in position to prove that every typable program is secure:

[1] Note that the case where $\langle S_1, E_1\rangle \neq^{\Gamma}_{\mathcal{L}} \langle R_2, F_2\rangle$ attends the case where only one of the computations performs an instant change and $\exists a_L \in E \cup F$.

Theorem 3.4 (Noninterference) *If P is typable in Γ then P is Γ-secure.*

Proof 3.4 *For any $\mathcal{L}$, define the relation $\mathcal{S}_\mathcal{L}$ on configurations $C_1 = \langle \Gamma_1, S_1, E_1, P_1\rangle$ and $C_2 = \langle \Gamma_2, S_2, E_2, P_2\rangle$, such that $C_1\ \mathcal{S}_\mathcal{L}\ C_2$ if and only if:*

- *P_i is typable in Γ_i for $i = 1, 2$, and*
- *$C_1 =_\mathcal{L}^{\Gamma_1\cap\Gamma_2} C_2$, and*
- *either*
  - *i. $P_i \in \mathcal{H}_{\mathsf{syn}}^{\Gamma_i,\mathcal{L}}$ for $i = 1, 2$, or*
  - *ii. $\langle \Gamma_1, P_1\rangle = \langle \Gamma_2, P_2\rangle$.*

Note first that if P is typable in Γ and $\langle S_1, E_1\rangle =_\mathcal{L}^\Gamma \langle S_2, E_2\rangle$, then $\langle \Gamma, S_1, E_1, P\rangle\ \mathcal{S}_\mathcal{L}\ \langle \Gamma, S_2, E_2, P\rangle$ by clause ii. Next we prove that $\mathcal{S}_\mathcal{L} \subseteq \approx_\mathcal{L}$. Suppose that $C_1\ \mathcal{S}_\mathcal{L}\ C_2$. Then we have $C_1 =_\mathcal{L}^{\Gamma_1\cap\Gamma_2} C_2$ by hypothesis, and either:

1. *Both $P_i \in \mathcal{H}_{\mathsf{syn}}^{\Gamma_i,\mathcal{L}}$ for $i = 1, 2$ by clause i. Since $\mathcal{H}_{\mathsf{syn}}^{\Gamma_i,\mathcal{L}} \subseteq \mathcal{H}_{\mathsf{sem}}^{\Gamma_i,\mathcal{L}}$ then $P_i \in \mathcal{H}_{\mathsf{sem}}^{\Gamma_i,\mathcal{L}}$.*
2. *$\langle \Gamma_1, P_1\rangle = \langle \Gamma_2, P_2\rangle$ by clause ii. We may assume that $P_i \notin \mathcal{H}_{\mathsf{syn}}^{\Gamma_i,\mathcal{L}}$, since otherwise we would fall into the previous point. If $C_1 \longmapsto C_1' = \langle \Gamma_1', S_1', E_1', P_1'\rangle$, then $P_1 \neq \mathtt{nil}$. Since $P_1 = P_2$, also $P_2 \neq \mathtt{nil}$ and by Theorem 2.1 $\exists! C_2'$ such that $C_2 \longmapsto C_2' = \langle \Gamma_2', S_2', E_2', P_2'\rangle$. By Theorem 3.3, we conclude that $C_1' =_\mathcal{L}^{\Gamma_1'\cap\Gamma_2'} C_2'$ and $\langle \Gamma_1', P_1'\rangle = \langle \Gamma_2', P_2'\rangle$. Hence, $C_1'\ \mathcal{S}_\mathcal{L}\ C_2'$ by clause ii.*

Let us now return to the program $P = \mathtt{if}\ x_H = 0\ \mathtt{then}\ \mathtt{nil}\ \mathtt{else}\ \mathtt{pause}$, so that we can understand the subtleties in this apparently straightforward definition of security. As can be seen from the definition of bisimulation, this program is accepted as secure for the reason that it is high. Let us argue why such programs do not pose security problems. Although we have not pursued this question formally here, our motivation stems from the following reasoning. Consider the program $P \upharpoonleft Q$, where Q would be an observer which we assume to be confined to the lowest level (and therefore typable, although the composition need not be). When control is given to Q, he is not able to know whether this is due to the suspension or to the termination of P. In particular, if Q starts by suspending itself, an instant change will occur in both cases. Unable to distinguish the two possible execution paths that P might have taken, Q cannot perform different actions accordingly. For similar reasons we do not consider necessary to require the synchronization of instant changes for matching two computations of a program.

4 Conclusion and related work

In this paper we have addressed the question of noninterference for reactive programs. We have presented a type system guaranteeing noninterference in a core imperative reactive language. We are currently studying a call-by-value language for mobility built around a reactive core, called ULM [3]. We intend to adapt to this language the techniques we have developed here.

As has been observed, reactive programs obey a fixed scheduling policy, which is enforced syntactically using the parallel construct $\upharpoonleft$. Other approaches to noninterference in the presence of scheduling include the probabilistic one, proposed for instance in [17] and [14]. In these papers scheduling is introduced at the semantic level (adding probabilities to the transitions), and security is formalized through a notion of probabilistic noninterference. It should be noted that, unlike [5], which allows to express different scheduling policies, and [14] which accounts for an arbitrary scheduler (satisfying some reasonable properties), here a fixed deterministic scheduling is in use. Indeed, the novelty of our work resides mainly in addressing the question of noninterference in a reactive scenario. The work [13] examines the impact of synchronization on information flow, and uses it as a means to study time leaks without explicitly introducing a scheduler. However the analogy cannot be pushed very far since [13] has no notion of instant (and thus no way of recovering from deadlocks) and uses asynchronous parallel composition.

References

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *IEEE*, 91(1):64–83, 2003.

[2] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. of Comput. Programming*, 19:87–152, 1992.

[3] G. Boudol. ULM, a core programming model for global computing. In *ESOP'04*, 2004.

[4] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *ICALP'01*, number 2076 in Lecture Notes in Computer Science, pages 382–395, 2001.

[5] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002.

[6] G. Boudol, I. Castellani, and A. Matos. Typing noninterference for reactive programs. Full paper available at "http://www-sop.inria.fr/mimosa/personnel/ana.matos/tec-rep2004.ps", 2004.

[7] F. Boussinot. Reactive Programming, Software available at "http://www-sop.inria.fr/mimosa/rp/", 2003.

[8] J.-F. Susini F. Boussinot. The sugarcubes tool box: a reactive JAVA framework. *Software Practice and Experience*, 28(14):1531–1550, 1998.

[9] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[10] N. Halbwachs. Synchronous programming of reactive systems, 1993.

[11] N. Yoshida K. Honda. A uniform type structure for secure information flow. In *Proceedings of the The 29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, 2002.

[12] A. Myers S. Zdancewic. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, 2003.

[13] A. Sabelfeld. The impact of synchronization on secure information flow in concurrent programs. In *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics*, 2001.

[14] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *13th Computer Security Foundations Workshop*. IEEE, 2000.

[15] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings POPL '98*, pages 355–364. ACM Press, 1998.

[16] Geoffrey Smith. A new type system for secure information flow. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 115–125. ACM Press, June 2001.

[17] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2-3), 1999.

[18] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.